The operand fields in the .OPT directive are only scanned for the first three characters. The individual .OPT operands are

1. SYM is used to control the printing of the symbol table at the end of the listing. The symbol table is not sorted. If NOSYM is selected, the symbol table is not printed.

2. ERR is used to control creation of error listing. To view only the errors on the assembly, use

   .OPT ERR,NOLIST

   When all the errors have been corrected, make another run using

   .OPT NOERR,LIST

3. LIST is used to control the generation of the listing which contains source input, errors & warnings, and and code generated. NOLIST suppresses the listing.

4. GEN is used to control printing of ASCII strings in the .BYTE directive. The first two characters only are printed if NOGEN is used. Further characters (normally two bytes per line) are printed if GEN is used.

5. TAB is used to control automatic spacing of labels, opcodes, and comments. Use of this option will save memory space in storing the source code, since only a single space is needed between labels, opcodes, and comments. The assembler output will be neatly formatted. With narrow-carriage terminals (for example, 32 characters/line CRTs), the spacing may be objectionable. Use of NOTAB suppresses the spacing.

Default settings for the .OPT directive are

```
.OPT      SYM, LIST, ERR, TAB
```

.END must be the last statement in a program and is used to signal the physical end of the text file. If no .END is used, the assembler returns you to the monitor. This is useful when doing multiple-file assembly. In this case, only the last file assembled must contain the .END directive.

ERROR MESSAGES

Error #1 Not Used

Error #2 FATAL - Label Previously Defined

The first field on the line is not an opcode, so it is interpreted as a label. If the current line is the first line in which that symbol appears as a label (or on the left side of an equals sign), it is put in the symbol table and tagged as defined in that line. However, if the symbol has appeared as a label, or on the left side of an equate, prior to the current line, the assembler finds the label already in the symbol table. The assembler does not allow redefinitions of symbols and will, in this case, print the error message.

Error #3 FATAL - Illegal Or Missing Opcode

The assembler searches a line until it finds the first non-blank character string. If this string is not one of the 55 valid opcodes, it assumes the string is a label and places it in the symbol table. It then continues parsing for the next non-blank string. If none is found, the next line will be read in and the assembly will continue. However, if a second field is found, it is assumed to be an opcode (since only one lable is allowed per line). If this character string is not a valid opcode, the error message is printed. This error can occur if opcodes are misspelled, in which case, the assembler will interpret the opcode as a label (if no label appears on the line). It will then try to assemble the next field as the opcode. If there is another field, this error will be printed. Check for more than one label on a line or a misspelled opcode.

Error #4 FATAL - Address Not Valid

An address referred to in an instruction or in one of the assembler directives (.BYTE and .WORD) is invalid. In the case of an instruction, the operand that is generated by the assembler must be greater than or equal to zero and less than or equal to $FFFF_{16}$ (2 bytes long). This excludes relative branches which are limited to ±127 bytes from the next instruction. If the operand generates more than two bytes of code, or is less than zero, this error message will be printed. For a .BYTE, each operand is limited to one byte, and for a .WORD, each operand is limited to two bytes. All operands must be greater than or equal to zero. This validity is checked after the operand is evaluated. Check for values of symbols used in the operand field (see the symbol table for this information).

Error #5 FATAL - Accumulator Mode Not Allowed

Following a legal opcode and one or more spaces is the letter A, followed by one or more spaces. The assembler is trying to use the accumulator (A means "accumulator mode") as the operand. However, the opcode in the statement is one which does not allow reference to the accumulator. Check for a statement labelled A (an illegal label) to which this statement is referring. If you were trying to reference the accumulator, look up the valid operands for the opcode used.

Error #6 FATAL - Forward Reference In .BYTE or .WORD

Error #7 FATAL - Ran Off End Of Line

This error message will occur if the assembler is looking for a needed field and runs off the end of the line before the field is found. The following should be checked for:

1. A valid opcode field without an operand field on the same line.

2. An opcode that was thought to take an implied operand which in fact required an operand.
3. An ASCII string that is missing the closing quote (be sure any embedded quotes are doubled - to have a quote at the end, there must be three quotes: 2 for the embedded quote and 1 to close off the string).
4. A comma at the end of the operand field indicates that there are more operands to come. If there aren't any other operands, the assembler will run off the line looking for them.

Error #8 — FATAL - Label Doesn't Begin With An Alphabetic Character.

The first non-blank field is not a valid opcode. Therefore the assembler tried to interpret it as a label. However, the first character of the field does not begin with an alphabetic character and the error message is printed. Check for an unlabelled statement with only an operand field that does start with a special character. Also check for illegal labels.

Error #9 — FATAL - Label Greater Than Six Characters

All symbols are limited to six characters in length. When parsing, the assembler looks for one of the separating characters to find the end of a label or string. If other than one of these separators is used, the error message will be printed - providing the illegal separator causes the symbol to extend beyond six characters in length. Check for no spacing between labels and opcodes. Also check for a comment line with a long first word that doesn't begin with a semi-colon. In this case, the assembler is trying to interpret part of the comment as a label.

Error #10 FATAL - Label Or Opcode Contains Non-Alphanumeric Character

Labels are made up of from one to six alphanumeric digits. The label field must be separated from the opcode field by one or more blanks. If a special character or other separator is between the label and the opcode, this error message might be printed.
The 55 valid opcodes are each three alphabetic characters. They must be separated from the operand field (if one is necessary) by one or more blanks. If the opcode ends with a special character (such as a comma), this error message will be printed.
In the case of a lone label or an opcode that needs no operand, they can be followed directly by a semicolon to denote the rest of the line as a comment.

Error #11 FATAL - Forward Reference In Equate Or Org

The expression on the right side of an equals sign contains a symbol that hasn't been defined previously. One of the operations of the assembler is to evaluate expressions or labels and assign addresses or values to them. The assembler processes the input values sequentially which means that all of the symbolic values that are encountered fall into two classes--already defined values and not previously encountered values. The assembler assigns defined values and builds a table of undefined values. When a previously used value is discovered, it is substituted into the table.
A label or expression which uses a yet undefined value is considered to be referenced forward to the to-be-defined value.
To allow for conformity of evaluating expressions, this assembler allows for one level of forward reference so that the following code is allowed:

Error #11 (Continued)

Line Number	Label	Opcode	Operand
100		BNE	New One
200	New One	LDA	#5

but the following is not allowed:

Line Number	Label	Opcode	Operand
100		BNE	New One
200	New One		Next + 5
300	Next	LDA	#5

This feature should not disturb the normal use of labels. The cure for this error

Line Number	Label	Opcode	Operand
100		BNE	New One
300	Next	LDA	#5
301	New One		Next + 5

is very simple and always solves the problem.

This error may also mean that the value on the right side of the = is not defined at all in the program in which case the cure is the same as for undefined values.

Due to the sequential processing of the assembler and the dependency of the value of the program counter on symbols, throughout the rest of the program, the assembler cannot process a forward reference in this type of statement. All expressions with symbols that appear on the right side of any equals sign must refer only to previously defined symbols for the equate to be processed.

Error #12 FATAL - Invalid Index. Index Must Be X Or Y

After finding a valid opcode, the assembler looks for the operand. In this case, the first character in the operand field is a left paren. The assembler interprets the next field as an indirect address which, with the exception of the jump statement, must be indexed by one of the index registers, X or Y. In the erroneous case, the character the assembler was trying to interpret as an index register

Error #12 (Continued) was not X or Y and the error was printed. Check for the operand field starting with a left paren. If it is supposed to be an indirect operand, recheck the format for the two types available. If the format was wrong (missing right paren or index register), this error will be printed. Also check for missing or wrong index registers in an indexed operand (form: expression, index register).

Error #13 FATAL - Invalid Expression In Operand

In evaluating an expression, the assembler found a character it couldn't interpret as being part of a valid expression. This can happen if the field following an opcode contains special characters not valid within expressions (e.g. parentheses). Check the operand field and make sure only valid special characters are within a field (between commas).

Error #14 FATAL - Undefined Assembler Directive

All assembler directives begin with a period. If a period is the first character in a non-blank field the assembler interprets the following character string as a directive. If the character string that follows is not a valid assembler directive, this error message will be printed. Check for a misspelled directive, or a period at the beginning of a field that is not a directive.

Error #15 FATAL - Invalid Operand For Page Zero Mode

Error #16 FATAL - Invalid Operand For Absolute Mode

Error #17 FATAL - Relative Branch Out Of Range

All of the branch instructions (excluding the two jumps) are assembled into 2 bytes of code. One byte is for the opcode and the other for the address to branch to. To allow a forward or backward branch, this branch is taken relative to the beginning of the next instruction, according to the address byte. If the value of the byte is 0-127 the branch is forward; if the value is 128-255 the branch is backward. (A negative branch is in 2's complement form). Therefore, a branch instruction can only branch forward or backward 127 bytes relative to the beginning of the next instruction. If an attempt is made to branch further than these limits, the error message will be printed.

Error #18 FATAL - Illegal Operand Type For This Instruction

After finding an opcode that does not have an implied operand, the assembler parses the operand field (the next non-blank field following the opcode) and determines what type of operand it is (indexed, absolute, etc.). If the type of operand found is not valid for the opcode, this error message will be printed. Check to see what types of operands are allowed for the opcode and make sure the form of the operand type is correct (see the section on addressing modes).

Error #19 FATAL - Out Of Bounds On Indirect Addressing

An indirect address is recognized by the assembler by the parentheses that surround it. If the field following an opcode has parens around it, the assembler will try to assemble

Error #19 (continued) it as an indirect address. Since indirects work only in page zero memory, if the address in the operand field is larger than 256 (one byte), this error message will be printed.

This error will only occur if the operand field is in correct form (i.e. an index register following the address), and the address field is out of page zero. To correct this, the address field must refer to page zero memory.

Error #20 FATAL - A, S, P, X, and Y Are Reserved Labels

A label on a statement is one of the five reserved names (A, X, Y, S AND P). They have special meaning to the assembler and therefore cannot be used as labels. Use of one of these names will cause the above error message to be printed and no code to be generated for the statement. The label does not get defined and will appear in the symbol table as an undefined variable. Reference to such a label elsewhere in the program will cause error messages to be printed as if the label were never declared.

Error #21 FATAL - Program Counter Negative! Reset To 0

An assembled program is loaded into core from position 0 to 64K (65536). This is the extent of the machine. Instructions can only refer to up to 2 bytes of information. Because there is not such a thing as negative memory, an attempt to reference a negative position will cause this error and the program counter (or pointer to the current memory location) will be reset to 0.
When this error occurs, the assembler continues assembling the code with the new value of the program counter. This

Error #20 (continued) could cause multiple bytes to be assembled into the same locations. Therefore, care is to be taken to keep the program counter within the proper limits.

Error #22 FATAL - Invalid Character. Expecting "=" For Org

Other error messages are mnemonic, such as BAD COM, for Bad Command, in the Editor. They are self-explanatory, and hence are not discussed herein.

APPENDIX A

Modifying The ARESCO Assembler For Two Pass Operation

Normal operation of the ARESCO Assembler/Text Editor allows the assembler portion of the package to assemble the source file into object code during a single reading of the file. The assembly is therefore fairly fast. The original version of the ARESCO Assembler was designed for the KIM-1, and it was an advantage to be able to assemble lengthy programs read from punched paper tape, without having to read the paper tape source code twice. In the version of the assembler designed for the APPLE II, this is no longer a benefit.

In order to assemble in a single pass, the assembler must build a symbol table and generate object code simultaneously. This works fine, except when the assembler is required to resolve forward references. (A forward reference is the use of an operand not previously defined, with the intention of defining the operand later in the program.) Consider this example:

```
10       LDY #04
20       CMP $43
30       BNE DONE
40       CLC
50 DONE  JMP $1C00
60 .END
```

When the assembler encounters the label DONE in line 30, it cannot calculate the relative branch offset, because it does not yet know the address of the label DONE. It handles this situation by reserving two bytes for the branch offset, marking the symbol table entry for DONE as an undefined reference. The notation "**" is printed in the listing of the assembled code. When DONE is defined in line 50, the assembler makes the cor-

rect entry in the symbol table, goes back and calculates the correct branch values for previous references to DONE, and inserts these values into the object code. The second byte reserved for the branch offset is set to EA (a NOP instruction).

If you assemble our example program, you will see that the branch address is listed as "**". If you examine the object code generated by the assembler, you will see that the correct value for the relative branch was automatically inserted into the object code, and is followed by an EA (NOP) instruction.

For most quick assemblies, this single-pass operation is simple and quite sufficient. For lengthy programs, however, a two-pass assembler may be needed, and it is easy to modify the ARESCO Assembler to provide the two-pass operation.

Modifications To The Assembler

Modify the following locations using the APPLE monitor:

```
257A: 4C F0 30
30F0: B1 52 A0 03 29 1F C9 10
30F8: D0 01 88 A9 01 4C 7D 25
3782: 20 11 37 4C 11 20
```

and save this modified version on disk or tape (2000.3800W).

Using The Modified (Two-Pass) Assembler

To use this new two-pass assembler, follow these simple steps:

1. Set the symbol pointers (1FDF-1FE2), as usual.
2. Enter the editor at 3743, as usual.
3. Input and correct the source code, as usual.
4. Assemble the source code, using the A command - and ignore any resulting error messages.
5. When the assembler exits to the APPLE monitor, restart the assembler by entering 3782G (3782 is the assembler "warm start" address). This will produce a correct listing and a proper object code. There will be no asterisks in the listing, and forward branches will be a single byte.

Note that when the one-pass assembler is in operation, the program counter is set to $200 if no "*=" statement is found. When using this modified, two-pass version of the assembler, the first line of the source code MUST be a definition of the program counter, such as

```
5 *=$200
```

The two-pass assembler will not work correctly without such a statement as the first line of code in the program.

Additional Modifications To The ARESCO Assembler/Text Editor

The ARESCO Assembler/Editor was originally designed to run on a KIM/TIM 6502 microcomputer. When the package was redesigned for the APPLE, massive page zero conflicts occurred. This problem was circumvented by keeping two copies of page zero in memory, one for use by the assembler/editor and one for use by the monitor I/O and disk routines. The assembler's page zero is stored in page 1F while the I/O and disk routines are being used, and the monitor's page zero is stored in page 1E while the assembler/editor is being operated. Locations 0300 through 0306 are used as temporary storage locations to preserve the values of A, X, Y, and P during I/O calls from the assembler/editor.

The MONASM routine located at #3711 saves the monitor's page zero and loads the assembler's page zero. The ASMMON routine at location $36DF saves the assembler's page zero and loads the monitor's page zero. All assembler/editor I/O routines begin with a call to ASMMON and end with a call to MONASM. Only the contents of register A are passed between the monitor and the assembler.

The following I/O routines are included in the assembler and may be disassembled, examined, and modified by the knowledgeable user: